ReCycle: Pipeline Adaptation to Tolerate Process Variation*

Abhishek Tiwari, Smruti R. Sarangi and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu
{atiwari,sarangi,torrellas}@cs.uiuc.edu

Abstract

Process variation affects processor pipelines by making some stages slower and others faster, therefore exacerbating pipeline unbalance. This reduces the frequency attainable by the pipeline. To improve performance, this paper proposes *ReCycle*, an architectural framework that comprehensively applies cycle time stealing to the pipeline — transferring the time slack of the faster stages to the slow ones by skewing clock arrival times to latching elements after fabrication. As a result, the pipeline can be clocked with a period close to the *average* stage delay rather than the longest one. In addition, ReCycle's frequency gains are enhanced with *Donor* stages, which are empty stages added to "donate" slack to the slow stages. Finally, ReCycle can also convert slack into power reductions.

For a 17FO4 pipeline, ReCycle increases the frequency by 12% and the application performance by 9% on average. Combining ReCycle and donor stages delivers improvements of 36% in frequency and 15% in performance on average, completely reclaiming the performance losses due to variation.

Categories and Subject Descriptors: B.8.0 [Hardware]: Performance and Reliability–*General*

General Terms: Performance, Design

Keywords: Pipeline, Process Variation, Clock Skew

1. Introduction

Process variation is a major obstacle to the continued scaling of integrated-circuit technology in the sub-45 nm regime. As transistor dimensions continue to shrink, it becomes successively harder to precisely control the fabrication process. As a result, different transistors in the same chip exhibit different values of parameters such as threshold voltage or effective channel length. These parameters in turn determine the switching speed and leakage of transistors, which are also subject to substantial fluctuation.

Variation in transistor switching speed is visible at the architectural level when it makes some unit in the processor too slow to meet timing, forcing the whole processor to operate at a lower frequency than nominal. Variation is already forcing designers to employ guard banding, and the margins are getting wider as technology scales. Bowman *et al*. suggest that variation may wipe out the performance gains of a full technology generation [6].

Careful timing design is especially important in state-of-the-art processor pipelines. The choices of what stages to have and what clock period they will all share are affected by many considerations [21, 23, 42]. With process variation, the logic paths in some stages become slower and those in other stages become faster after fabrication, exacerbating pipeline unbalance and reducing the frequency attainable by the pipeline.

Current solutions to the general problem of process variation can be broadly classified into circuit-level and architecture-level techniques. At the circuit level, there are multiple proposed techniques, including adaptive body biasing (ABB) [45] and adaptive supply voltage (ASV) scaling [9]. Such techniques are effective in many cases, although they add complexity to the manufacturing process and have other side effects. Specifically, boosting frequency with ABB increases leakage power and doing it with ASV can have a damaging effect on lifetime reliability.

Architecture-level techniques are complementary to circuit-level ones. However, most of the ones proposed so far target a small number of functional blocks, namely the register file and execute units [31] and the data caches [34]. Other techniques have focused on redesigning the latching elements [17, 46]. These techniques likely involve a substantial design effort and hardware overhead.

In this paper, we propose to tolerate the effect of process variation on processor pipelines with an architecture-level technique that: (i) does not adversely affect leakage or hardware reliabiliy, (ii) is globally applicable to all subsystems in a pipeline, and (iii) has a negligible hardware overhead. It is based on the comprehensive application of cycle time stealing [4], where the time slack of the faster stages in the pipeline is transferred to the slower ones by skewing the clock arrival times to latching elements. As a result, the pipeline can be clocked with a period close to the *average* stage delay rather than the longest one. We call this approach *ReCycle* because the time slack that was formerly wasted by the faster stages is now "recycled" to the slower ones.

We show that ReCycle increases the frequency of the pipeline without changing the pipeline structure, pipeline depth, or the inherent switching speed of transistors. Such increase is relatively higher the deeper the pipeline is. Moreover, ReCycle can be combined with *Donor* pipeline stages, which are empty stages added to the critical loop in the pipeline to "donate" slack to the slow stages, enabling an increase in the pipeline frequency. We can also use

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Smruti R. Sarangi is now with Synopsys, Bangalore, India. His email is ssarangi@synopsys.com

ISCA'07, June 9–13, 2007, San Diego, California, USA.

ReCycle to push the slack of non-critical pipeline loops to their feedback paths, and then consume it there to reduce wire power or to improve wire routability. Finally, ReCycle can also be used to salvage chips that would otherwise be rejected due to variation-induced hold-time failures.

Our evaluation compares variation-affected pipelines without and with ReCycle. On average for a 17FO4 pipeline, ReCycle increases the frequency by 12%, thereby recovering 63% of the frequency lost to variation, and speeding up our applications by 9%. Combining ReCycle and donor stages is even more effective. Compared to the pipeline without ReCycle, it increases the frequency by 36% and the performance by 15% on average, performing even better than a pipeline without process variation. Finally, ReCycle also saves 7-15% of the power in feedback paths.

This paper is organized as follows. Section 2 gives a background; Sections 3, 4, and 5 present ReCycle's ideas, uses, and implementation, respectively. Sections 6 and 7 evaluate ReCycle; and Section 8 discusses related work.

2. Background

2.1. Pipeline Clocking

One of the most challenging tasks in pipeline design is to ensure that the pipeline is clocked correctly. Data propagation delay and clock period have to be such that, in each latch element, the setup (T_{setup}) and hold (T_{hold}) times are maintained.

Often, it is desired to fit more logic in a pipeline stage than the cycle time would allow. This can be accomplished without changing the pipeline frequency by using a technique called *Cycle Time Stealing* [4]. With this technique, a stage utilizes a portion of the time allotted to its successor or predecessor stages. This forcible removal of time from another stage is typically obtained by adjusting the clock arrival times.

Consider a pipeline stage that is preceded by flip-flop FF_i (for initial) and followed by flip-flop FF_f (for final). The stage can steal time from its successor stage by delaying the clocking of FF_f by a certain time or skew δ_f. Similarly, it can steal time from its predecessor stage by changing the clocking of FF_i by a skew δ_i that is negative. In all cases, since we do not change the cycle time, one or more stages have to have at least as much slack as the amount stolen.

Under cycle time stealing, the setup and hold constraints still have to be satisfied. Assume that the data propagation delay in the stage is T_{delay} and the pipeline's clock period is T_{CP}. The data generated at FF_i by a clock edge must arrive at FF_f no later than the setup time before the arrival of the next clock edge at FF_f (Equation 1). Moreover, the data generated at FF_i by a clock edge must arrive at FF_f no sooner than the hold time after the arrival of the clock edge at FF_f (Equation 2).

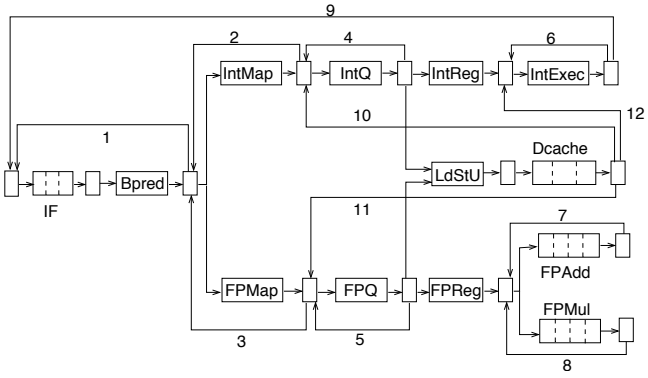
$$\delta_i + T_{delay} + T_{setup} \leq T_{CP} + \delta_f \quad (1)$$

$$\delta_i + T_{delay} \geq \delta_f + T_{hold} \quad (2)$$

2.2. Pipeline Loops

Pipeline loops are communication loops that appear when the result of one stage is needed in the same or an earlier stage of the pipeline [5, 10]. Loops are caused by data, control, or structural hazards. A loop is typically composed of one or more pipeline stages and a feedback path that connects the end stage to the begin stage.

As an example, Figure 1(a) shows a simplified version of the Alpha 21264 pipeline [28] that we use to demonstrate ReCycle. The figure does not show the physical structure of the pipeline. Rather, it shows a logical structure. Each long box represents a logical stage, while short boxes are pipeline registers between them. Some logical stages are broken down into multiple physical stages, as shown with dashed lines. Lines between logical stages represent communication links.



(a) Simplified logical pipeline structure.

Name	Description	Fdbk Path	Components
Fetch	Dependence between PC and Next PC	1	IF, Bpred, 1
Int rename	Dependence between inst. assigning a rename tag and a later one reading the tag	2	IntMap, 2
FP rename		3	FPMap, 3
Int issue	Dependence between the select of a producer inst. and the wakeup of a consumer	4	IntQ, 4
FP issue		5	FPQ, 5
Int ALU	Forwarding from execute to execute	6	IntExec, 6
FPAdd		7	FPAdd, 7
FPMul		8	FPMul, 8
Branch mispred.	Mispredicted branch	9	IF, Bpred, IntMap IntQ, IntReg, IntExec, 9
Int load misspecul	Load miss replay	10	IntQ, LdStU, Dcache, 10
FP load misspecul		11	FPQ, LdStU, Dcache, 11
Load forward	Forwarding from load to integer execute	12	IntExec, 9, IF, Bpred, IntMap, IntQ, LdStU, Dcache, 12

(b) Pipeline loops.

Figure 1: Simplified version of the Alpha 21264 pipeline used to demonstrate ReCycle: logical pipeline structure (a) and pipeline loops (b).

The figure depicts the front-end stages and then, from top to bottom, the stages in the integer datapath, load-store unit and cache, and floating-point datapath. While the real processor has more communication links, we only show those that we consider most important or most time critical. For example, we do not show the write back links, since write back is less time critical. The feedback paths are labeled.

Figure 1(b) describes the pipeline loops in the simplified pipeline. The first two columns name and describe, respectively, the loop. The next two columns show the feedback path that creates the loop and the components of the loop.

Note that our loops are not exactly the same as those in [5, 10]. Here, we examine a more complicated pipeline, and have not shown

all the communication links. In particular, we only show the feedback paths that we consider most important. For example, we do not show all the forwarding paths. While we will base our analysis on this simplified pipeline, ReCycle is general enough to be applicable to more complicated pipelines.

2.3. Process Variation and Its Impact

While process variation exists at several levels, we focus on Within-Die (WID) variation, which is caused by both *systematic* effects due to lithographic irregularities and *random* effects primarily due to varying dopant concentrations [43]. Systematic variation exhibits strong spatial correlation — structures that are close together are likely to have similar values — while random variation does not.

Two important process parameters affected by variation are the threshold voltage (V_t) and the effective channel length (L_{eff}). Variation of these parameters directly affects a gate's delay (T_g), as given by the alpha-power model [38]:

$$T_g \propto \frac{L_{eff} V_{dd}}{\mu (V_{dd} - V_t)^{\alpha}} \quad (3)$$

where μ is the carrier mobility, V_{dd} is the supply voltage, and α is usually 1.3. Both μ and V_t are a function of the temperature *T*.

We treat random and systematic variation separately. We model the systematic variation of V_t with a multivariate normal distribution with a specific correlation structure [43]. It is characterized by three parameters: μ, σ_{sys}, and ϕ. Specifically, we divide the chip into a grid with 1M cells. Each cell takes on a single value of V_t as given by a multivariate normal distribution with parameters μ and σ_{sys}. Along with this, V_t is spatially correlated.

We assume that the correlation is isotropic and independent of position [47]. This means that the correlation between two points $\vec{x}$ and $\vec{y}$ in the grid depends on the distance between them and not on the direction or position. Consequently, we express the correlation function of $V_t(\vec{x})$ and $V_t(\vec{y})$ as $\rho(r)$, where $r = |\vec{x} - \vec{y}|$. By definition, $\rho(0)$=1 (i.e., totally correlated). We also set $\rho(\infty)$=0 (i.e., totally uncorrelated). We then assume that $\rho(r)$ changes with *r* as per the Spherical distribution [12]. In the Spherical distribution, $\rho(r)$ decreases from 1 to 0 smoothly and reaches 0 at a distance ϕ called *range*. Intuitively, this means that at distance ϕ, there is no significant correlation between the V_t of two transistors. This approach matches empirical data obtained by Friedberg *et al*. [19]. ϕ is given as a fraction of the chip's width.

Random variation of V_t occurs at a much finer granularity than systematic variation: it occurs at the level of individual transistors. We model it as an uncorrelated normal distribution with σ_{rand} and a zero mean.

L_{eff} is modeled like V_t with a different μ, σ_{sys}, and σ_{rand} but the same ϕ.

From the V_t and L_{eff} variation, we compute the T_g variation using Equation 3. We then use the critical path model of Bowman *et al*. [6] to estimate the frequency supported by each pipeline stage. This model takes the number of gates (n_{cp}) in a critical path and the number of critical paths (N_{cp}) in a structure, and computes the probability distribution of the longest critical path delay ($\max\{T_{\text{cp}}\}$) in the structure. This is the path that determines the maximum frequency of the structure, which we set to be $1/\max\{T_{\text{cp}}\}$.

For simplicity, we model a critical path as n_{cp} FO4 gates connected by very short wires — where n_{cp} is the useful logic depth of a pipeline stage. Unfortunately, accurately estimating N_{cp} for each pipeline stage is difficult because N_{cp} is design-specific and not publicly available for a design. Consequently, we assume that critical paths are distributed in a spatially-uniform manner on the processor layout — except in the L2, whose paths we assume never affect the cycle time. From the layout area of each pipeline stage and Bowman *et al*.'s estimate that a high-performance processor chip at our technology node has about 10,000 critical paths [6], we determine the critical paths in each stage. The slowest critical path in a stage determines the frequency of the stage; the slowest stage determines the pipeline frequency.

3. Pipeline Adaptation with ReCycle

3.1. Main Idea

To understand the idea behind ReCycle, consider the pipeline of Figure 2(a) and call T_i the time taken to perform the work in stage *i*. For simplicity, in the absence of process variation, we assume that T_i is the same for all *i* and, therefore, the pipeline's period is $T_{CP} = T_i, \forall i$. When variation sets in, it slows down some stages while it speeds up others. As shown in Figure 2(a), the resulting unbalanced pipeline has to be clocked with a longer period $T_{CP} = Max(T_i), \forall i$.

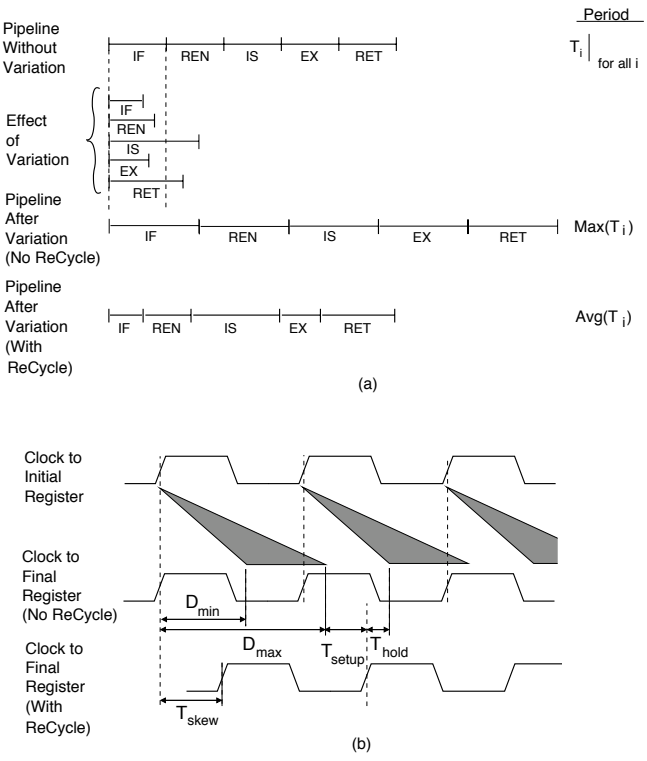


Figure 2: Effect of process variation on pipelines (a) and skewing a clock signal (b).

With ReCycle, we comprehensively apply cycle time stealing to correct this variation-induced pipeline unbalance. The resulting clock period of the pipeline is $T_{CP} = Average(T_i), \forall i$. This period can potentially be similar to that of the no-variation pipeline. As shown in Figure 2(a), the slow stages get more than a clock period to propagate their signal, at the expense of faster stages that transfer their slack.

With this approach, we do not need to change the pipeline structure, pipeline depth, or the inherent switching speed of transistors.

Figure 2(b) depicts the timing diagram for a slow stage, showing the clock signal to its initial pipeline register and to its final register (the latter without and with ReCycle). Data propagation in the stage can take a range of delays (D_{min}, D_{max}), depending on which path it uses. This range is shown as a shaded cone. Without ReCycle, the figure shows that the signal may take too long to be latched by the final register.

With ReCycle, the clock of the final register is delayed by T_{skew}. T_{skew} is chosen so that, even if the signal takes D_{max}, it reaches the final register early enough to satisfy the setup time (T_{setup}) (Figure 2(b)). Since the clock period is smaller than D_{max}, two signals can simultaneously exist in the logic of one stage in a wave-pipelined manner [11]. This can be seen by the fact that the cones of two signals overlap in time. In addition, the minimum delay D_{min} has to be long enough so that the hold time (T_{hold}) of the final register is satisfied (Figure 2(b)).

In general, we will skew the clocks of both the initial and final registers of the stage. As per Section 2.1, we call such skews δ_i and δ_f, respectively. Consequently, T_{skew} in Figure 2(b) is δ_f-δ_i. In a slow stage, $\delta_f > \delta_i$; in a fast one, $\delta_f < \delta_i$. For ReCycle to work, all the stages in the pipeline have to satisfy the setup and hold constraints of Equations 1 and 2 which, expressed in terms of D_{min} and D_{max} can be rewritten as:

$$\delta_f - \delta_i + T_{CP} \geq D_{max} + T_{setup} \quad (4)$$

$$\delta_f - \delta_i \leq D_{min} - T_{hold} \quad (5)$$

In a real pipeline, this simple model gets complicated by the fact that a pipeline is not a single linear chain of stages. Instead, as shown in Figure 1(a), the pipeline forks to generate subpipelines (e.g., the integer and floating-point pipelines) and loops back to previous stages through feedback paths (e.g., the branch misprediction loop).

With ReCycle, stages can only trade slack if they participate in a *common loop*. As an example, in Figure 1(a), the IntExec and the Bpred stages can trade slack because they belong to the branch misprediction loop. However, the IntExec and the FPAdd stages cannot trade slack.

3.2. Finding the Optimal Period and Skews

Given an arbitrary pipeline, we would like to find the shortest clock period T_{CP} that we can clock it at, and the set of time skews δ that we need to apply to the different pipeline registers to make that possible. The setup and hold constraints of Equations 4 and 5 are linear inequalities. Consequently, the problem of finding the optimal period and skews can be formulated as a linear program, where we are minimizing T_{CP} subject to the setup and hold constraints for all the stages in the pipeline.

In this linear program, the unknowns are T_{CP} and the skews (δ_i and δ_f) of the initial and final pipeline registers of each individual stage. Such skews can take positive or negative values. The known quantities are the delays of the slowest and fastest paths (D_{max} and D_{min}) in each pipeline stage, and the setup and hold times (T_{setup} and T_{hold}) of each pipeline register. We will see later how D_{max} and D_{min} can be estimated.

To solve this linear program, we can use a conventional algorithm, which typically runs in asymptotically exponential time. Here, instead, we choose to map this problem to a graph, where nodes represent pipeline register skews and the directed edges represent the setup and hold constraints. The representation for a stage is shown in Figure 3, where the edge values are additive to the node values. We represent the whole pipeline as a graph in this way. With this representation, we can solve the problem of finding the optimal skew assignment using a shortest-paths algorithm proposed by Albrecht *et al*. [3].

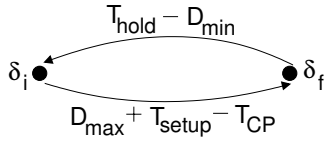


Figure 3: Constraint graph.

This algorithm runs in worst-case asymptotic time O(NumEdges $\times$ NumNodes + NumNodes$^2 \times$ log(NumNodes)) and is much faster in practice. To determine an upper bound on the execution time of this algorithm, let us consider the Bellman-Ford algorithm (BF), which is a less efficient shortest-paths algorithm. An invocation of the BF algorithm iterates over all the nodes in the graph. In each iteration, it relaxes all graph edges. Relaxing an edge involves 3 loads, 2 integer ALU operations, and 1 store. Consequently, a BF invocation involves 4$\times$NumNodes$\times$NumEdges memory accesses and 2$\times$NumNodes$\times$NumEdges integer ALU operations. Since, in practice, only 2 calls to BF are required to converge for this type of problem, the total number of operations is twice that. For our model of the Alpha 21264 pipeline (Section 2.2), there are 14 nodes and 26 edges, which brings the total number of memory accesses to $\approx$2,900 and integer ALU operations to $\approx$1,500. Memory accesses have high locality because they only read and write the nodes and edges. Overall, the execution takes little time. In the rest of the paper, we will refer to Albrecht *et al*.'s algorithm as the ReCycle algorithm.

The advantage of using this algorithm is two-fold. First, it is much faster than conventional linear programming approaches. Second, it identifies the loop that limits any further decrease in T_{CP}, namely the *critical* loop. Overall, after applying this algorithm, we obtain three results: (i) the shortest clock period T_{CP} that is compatible with all the constraints, (ii) the individual clock skew δ to apply to each pipeline register, and (iii) the critical pipeline loop.

3.3. Applying ReCycle

Recycle applies cycle time stealing [4] in a comprehensive manner to compensate for process variation in a pipeline. It relies on tunable delay buffers in the clock network that enable the insertion of intentional skew to the signal that reaches individual pipeline registers. We will outline an implementation of such buffers in Section 5.1.

To determine the skews to apply, we need to estimate the maximum (D_{max}) and minimum (D_{min}) delay of each stage. For a given stage, these parameters can be approximately obtained as follows. At design time, designers should identify two groups of paths: those that will contain the slowest one and those that will contain the fastest one. This can be done with timing analysis tools plus the addition of a guard band to take into account the effects of the expected systematic variation after fabrication — note that random variation is typically less important, since its effects on the gates of a path tend to cancel each other. In addition, designers should construct a few BIST vectors that exercise these paths.

After fabrication, the processor should be exercised with these BIST vectors at a range of frequencies. From when the test fails, designers should be able to identify the actual fastest and slowest

paths under these conditions, and D_{max} and D_{min}. Since the testing of each stage can proceed in parallel, characterization of the entire pipeline can be done quickly.

Note that the application of ReCycle does not assume that the pipeline stages were completely balanced before variation. In reality, pipelines are typically unbalanced. Since ReCycle can leverage unbalance irrespective of its source, the more unbalance that exists before variation, the higher the potential benefits of ReCycle. In reality, however, some of the unbalance detected at design time will have been eliminated by introducing various time-borrowing circuits in the design. ReCycle is compatible with the existence of such circuits, and will still exploit the variation-induced unbalance.

ReCycle can be applied once by the chip manufacturer after the chip is fabricated. After determining the delays, the manufacturer runs the algorithm of Section 3.2 to determine the skews, and programs the latter in the delay buffers. The chip is then run at the chosen T_{CP}. Note that operating the chip at lower frequencies is still possible, since the setup and hold constraints for all pipeline registers would still be satisfied.

In addition, we can envision automatically applying ReCycle dynamically, as chip conditions such as temperature change. Such ability requires embedding circuitry to detect changes in path delays, such as ring oscillators, temperature sensors, delay chains or flip-flop modifications [1, 15]. Once the delays are known, our algorithm of Section 3.2 can determine the optimal T_{CP} and the skews very quickly. Specifically, as indicated in Section 3.2, our algorithm requires ≈4,400 basic operations for our model of the Alpha 21264 pipeline — which can be performed in about the same number of cycles.

4. Using ReCycle

ReCycle has several architectural uses that we outline here.

4.1. Enabling High-Frequency, Long Pipelines

The basic use of ReCycle is to enable high-frequency, long pipelines. With process variation, the transistors in one or several stages of a long pipeline are likely to be significantly slower than those in other stages. Without ReCycle, these stages directly limit the pipeline frequency; with ReCycle, the delay in these stages is averaged out with that of fast stages. With more stages in long pipelines, the variations in stage delays average out more effectively.

While Section 7.2 presents simulations that support this conjecture, this section introduces a simple, intuitive analytical model that gives insight into this issue. Specifically, consider a linear pipeline with *N* stages. For *this model only*, assume that (i) in each pipeline stage, all paths have the same delay, (ii) across stages, such delay is uncorrelated, and (iii) the delay is normally distributed with mean μ and standard deviation σ. Moreover, for simplicity, assume also that T_{setup} and T_{hold} are zero.

Denote the path delays in each stage as T_1, T_2, ..., T_N. The cumulative distribution function of the pipeline's clock period ($F_{CP}(x)$) is the probability that the pipeline can cycle with a period smaller than or equal to a given value ($P(T_{CP} \leq x)$).

For a pipeline without ReCycle, such cumulative distribution function is:

$$F^{nr}_{CP}(x) = P(T_{CP} \leq x) = P(T_1 \leq x \cap \ldots \cap T_N \leq x)$$

Given that we assume that path delays are independent across stages,

$$F^{nr}_{CP}(x) = P(T_{CP} \leq x) = P(T_1 \leq x) \times \ldots \times P(T_N \leq x)$$

If we call $F(x)$ the cumulative distribution function of the path delay in a stage, given that all stages have the same distribution, we have:

$$F^{nr}_{CP}(x) = P(T_{CP} \leq x) = F(x) \times \ldots \times F(x) = (F(x))^N$$

In a pipeline with Recycle, the delay of a stage can be redistributed to other stages, and the pipeline's period is given by the average of the stage delays. Specifically, the cumulative distribution function of the pipeline's clock period is:

$$F^{r}_{CP}(x) = P(T_{CP} \leq x) = P(\frac{T_1 + \ldots T_N}{N} \leq x) = F(x)$$

The last equality used the fact that the average of *N* independent random variables distributed normally with μ and σ is a random variable distributed normally with the same μ and σ.

From these equations, we see that $F^{nr}_{CP}(x) = (F^{r}_{CP}(x))^N$, where $F^{r}_{CP}(x) = F(x) < 1$. This allows us to draw an important conclusion: as we add more stages to the pipeline (*N* increases), the pipeline with ReCycle performs *exponentially* better than the one without it — i.e., the relative ability of ReCycle to make timing improves exponentially with pipeline depth.

4.2. Adding *Donor* Stages

A second use of ReCycle is to increase the frequency of a pipeline further by adding *Donor* pipeline stages. A donor stage is an empty stage that is added to the critical loop of the pipeline — i.e., the loop that determines the cycle time of the pipeline. The donor stage introduces additional slack that it "donates" to the other stages in the critical loop. This enables a reduction in the pipeline's clock period.

Donor stages are supported by including an additional pipeline register immediately after the output pipeline register of some pipeline stages. We call such registers *Duplicates*. In normal operation, a duplicate register is transparent and, as described in [33], introduces minor time overhead. To insert a donor stage after a stage, we enable its duplicate register. In our experiments, we add one duplicate register to each of the 13 logical pipeline stages in the Alpha pipeline of Figure 1(a). In this way, we ensure we cover all the pipeline loops.

Adding an extra stage to the pipeline incurs an IPC penalty, so it must be carefully done to deliver a net positive performance improvement. To select what donor stage(s) to add, we need to have a way of measuring their individual impact on the IPC of the applications. Then, we choose the one(s) that deliver the highest performance. The selection algorithm that we use is called the Donor algorithm.

The Donor algorithm proceeds as follows. Given an individual pipeline, we run the ReCycle algorithm to identify the critical loop. Then, we select one duplicate register from the critical loop and create a donor stage, rerun the ReCycle algorithm to set the new time skews and clock period, and measure the IPC. We repeat this process for all the duplicate registers in the loop, one at a time. The donor stage that results in the highest performance is accepted. After this, we run the ReCycle algorithm again to identify the new critical loop and repeat the process on this loop. This iterative process can be repeated until the pipeline reaches the power limit.

The Donor algorithm can be run statically at the manufacturer's site once or dynamically at runtime many times. In the former case, the manufacturer has a representative workload, and makes each decision in the algorithm based on the impact on the performance of the workload.

If the Donor algorithm is run dynamically, we rely on a phase detector and predictor (e.g., [40]) to detect phases in the running application. At the beginning of each new phase, the system runs the whole algorithm to decide what donor stages to add. The algorithm overhead is tolerable because application phases are long — the average phase is typically over 100ms. Moreover, during the period needed to profile the IPC of a given pipeline configuration (e.g., ≈10,000 cycles), the application is still running.

Note, however, that at every step in the Donor algorithm that we want to change the clock skews in the pipeline, we need to wait until the pipeline drains. Consequently, such operation is as expensive as a costly branch misprediction. To reduce overheads, since program phases tend to repeat, we envision that, after the system selects the skews, period, and donor(s) for a new phase, it saves them in a table. The data in the table will be reused if the phase is seen again in the future. Moreover, as an option, we may decide to stop after we have added one or two donor stages.

Supporting the ability to add donor stages necessarily complicates the pipeline implementation. For example, extending the number of cycles taken by a functional unit introduces complexity in the instruction scheduler. We are not aware of any work on systematically managing variable numbers of cycles for logical pipeline stages — although some restricted schemes have been recently proposed [34] (Section 8). We plan to target this problem in our future work.

4.3. Pushing Slack to Feedback Paths

A third use of ReCycle is to push the slack of non-critical loops to the loops' feedback paths. Such slack can then be used to reduce power or to improve wire routability. To see why, recall that the pipeline model that we are using (Section 2.2) models loops as sets of stages with feedback paths. The latter are abstracted away as one-cycle stages of simply wires with repeaters. Repeaters are typically inverters that, by interrupting long wires, reduce the total wire delay [22].

Two loops in a pipeline can be disjoint or overlapping. For example, Figure 4 shows two overlapping loops from Figure 1(a): the branch misprediction one and the integer load misspeculation one.

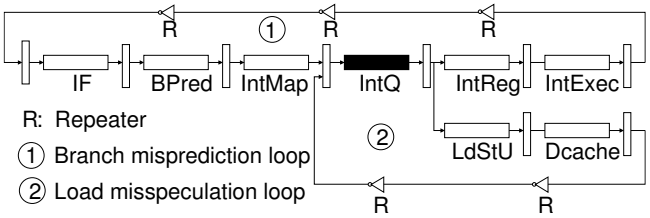


Figure 4: Example of overlapping loops.

In all the pipeline loops but the critical one, we use ReCycle to push all the slack in the loop to its feedback path. This does not affect the cycle time. Note that a stage that belongs to multiple loops has a special property: its slack is transferred to the feedback paths of *all* the loops it belongs to. For example, in Figure 4, the slack in the *IntQ* stage is passed simultaneously to both feedback paths.

By accumulating the slacks in the feedback paths, we can perform the following two optimizations.

4.3.1. Power Reduction

With optimal repeater design, about 50% of the power in a feedback path is dissipated in the repeaters [26]. Eliminating repeaters would save power, but it would also increase the delay of the feedback path, since a wire delay is $D = kl^2$, where l is the length of the wire without repeaters. Consequently, we propose to save power by eliminating as many repeaters as it takes to consume all the slack in the feedback path.

We envision an environment where the manufacturer, after measuring the effect of process variation on a particular pipeline, could eliminate individual repeaters from feedback paths to save power. In this case, we would proceed by removing one repeater at a time, selecting first repeaters between adjacent shortest wire segments (l_s). If we assume a wire with repeaters designed for optimal total delay, the delay through a repeater is equal to the delay through a wire segment [22]. Consequently, eliminating one repeater increases the delay from $3kl_s^2$ to $k(2l_s)^2$, which is kl_s^2.

4.3.2. Improved Routability

The slack of the feedback paths can instead be used to ease wire routing during the layout stage of pipeline design. Specifically, we can give the routing tool more flexibility to either lengthen the wires or put them in slower metal layers. Unfortunately, the routing stage is pre-fabrication and, therefore, we do not know the exact slack that will be available for each feedback path after fabrication. Consequently, the amount of leeway given to the routing tool has to be based on statistical estimates. We can use heuristics such as giving leeway only to the feedback paths of loops that are long — since they are unlikely to be critical because they can collect slack from many stages — and giving no leeway to the feedback paths of the loops that are very short — since one of them is likely to be the critical loop. In any case, even if for a particular pipeline, a loop whose feedback path was routed suboptimally ends up being the critical loop, we have not hurt correctness: ReCycle will simply choose a slighly longer clock period than it would have chosen otherwise.

We lack the infrastructure to properly evaluate this optimization. However, discussions with Synopsys designers suggest that the leeway that ReCycle provides would ease the job of routing the feedback paths of the pipeline.

4.4. Salvaging Chips Rejected Due to Hold Violations

A final use of ReCycle is to salvage chips that would otherwise be rejected due to variation-induced hold-time failures. This is a special case of ReCycle's use of cycle time stealing to improve pipelines after fabrication. However, while correcting setup violations (violations of Equation 4) can be accomplished through other, non-ReCycle techniques, correcting hold violations (violations of Equation 5) after fabrication with other techiques is harder. Specifically, a setup-time problem can be corrected by increasing the pipeline's clock period. However, correcting a hold-time problem after fabrication can be done only with trickier techniques such as slowing down critical paths by decreasing the voltage — with an adverse effect on noise margins. As a result, chips with hold-time problems typically end up being discarded.

The ReCycle framework seamlessly fixes pipelines with hold failures. Referring to Equation 5, a hold failure makes the right side negative for some pipeline stage, but ReCycle can make the left side negative as well. Running the ReCycle algorithm of Section 3.2

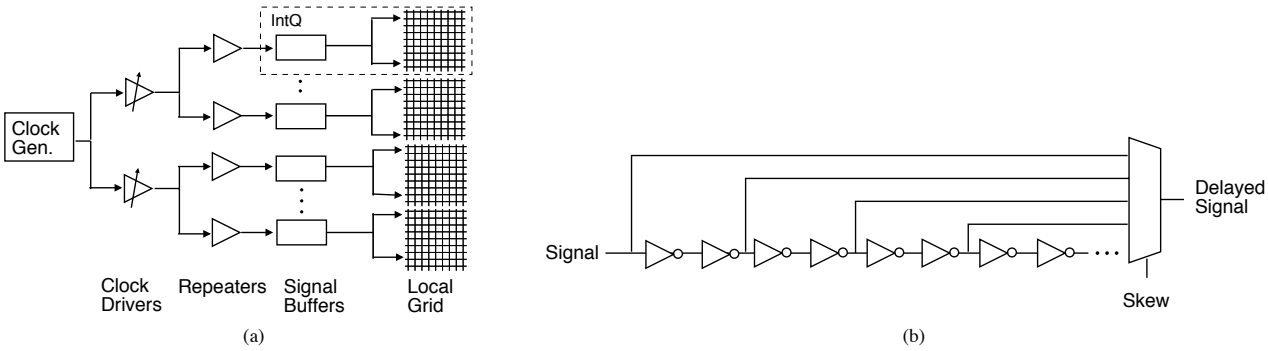


Figure 5: Skewing the clock signal: clock distribution network (a) and circuitry to change the delay of the signal (b).

will compute the optimal register skews for all stages to make such a pipeline reusable.

5. Implementation Issues

ReCycle has three components: tunable delay buffers, the software system manager, and duplicate registers. In addition, it can optionally have a phase detector and predictor, and temperature sensors. In this section, we overview their implementation and then show the overall ReCycle system.

5.1. Tunable Delay Buffers

ReCycle uses Tunable Delay Buffers (TDB) in the clock network to intentionally skew the signal that reaches individual pipeline registers. This can be easily done. Figure 5(a) shows a conventional clock network, where the clock signal is distributed through a multi-level network — usually a balanced H tree. The signal driven by the clock generator is boosted by repeaters and buffered in signal buffers — at least once, but often at a few levels — before driving a local clock grid. A local clock grid clocks a pipeline stage. This multi-level tree is strategically partitioned into zones that follow pipeline-stage boundaries.

We replace the last signal buffer at each zonal level in Figure 5(a) with a TDB, capable of injecting an intentional skew into its clocking subtree. This can be done by simply adding a circuit to delay the clock signal, for example as shown in Figure 5(b). A string of inverters is tapped into at different points to sample the signal at different intervals, and then a multiplexer is used to select the signal with the desired delay. A similar design is used in the Itanium clock network [14] — in their case to ensure that all signals reach the stages with the same skew.

The TDB itself could be subject to variation. This can be avoided by sizing its transistors larger.

5.2. System Manager

We propose to implement the ReCycle algorithm in a privileged software handler that executes below the operating system like the System Manager (SM) in Pentium 4 [39]. The ReCycle algorithm code and its data structures are stored in the SM RAM. When a System Management Interrupt (SMI) is generated, the ReCycle SM handler is invoked. The handler determines the new pipeline register skews and programs them into the TDBs. It also determines the new cycle time. As indicated in Section 3.2, the ReCycle algorithm performs about 4,400 basic operations for our pipeline, which take around 750ns on a 6GHz processor.

An SMI can be generated in two cases: when chip conditions such as temperature change (Section 3.3) or, if donor stages are supported, when the currently running application enters a new phase (Section 4.2). In the former case, the SMI is generated by sensors that detect when path delays change, such as a temperature sensor. In the latter case, the SMI is generated by a phase detector and predictor, such as the hardware unit proposed by Sherwood *et al*. [40].

5.3. Duplicate Registers

To apply the Donor Stage optimization of Section 4.2, we include one duplicate register in each *logical* pipeline stage — for example, immediately after the output pipeline register of its last physical stage. By default, these duplicate registers are disabled; when one is enabled, it creates a donor stage.

Previous work on variable pipeline-depth implementations shows how pipeline registers can be made transparent using pass-transistor multiplexing structures [33]. In our design, the single-bit enable/disable signals of all duplicate registers are collected in a special hardware register called Donor Creation register. Such register is set in privileged mode by the ReCycle SM handler.

5.4. Overall ReCycle System

The overall ReCycle system is shown in Figure 6. The figure shows one logical stage comprised of one physical stage. The duplicate register of the previous stage (shown in dashed lines) is not enabled, but the one of this stage is.

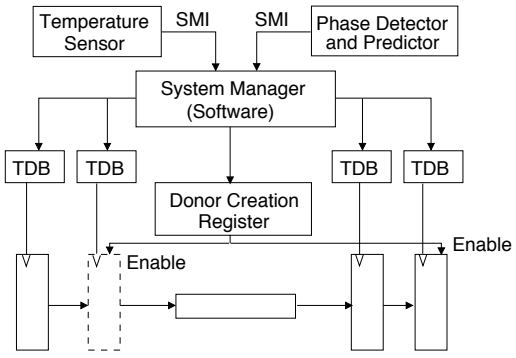


Figure 6: Overall ReCycle system.

The hardware overhead of ReCycle is as follows. For each logical stage, ReCycle adds a duplicate pipeline register and its TDB. A TDB is a signal buffer like those in conventional pipelines augmented with a chain of inverters and a multiplexer. Moreover, for each physical pipeline stage, ReCycle augments the existing clock signal buffer with a chain of inverters and a multiplexer. Finally, ReCycle adds the Donor Creation register. Optionally, ReCycle also uses a phase detector and predictor, and temperature sensors. Section 6.2 quantifies these resources for the actual pipeline modeled.

6. Evaluation Setup

6.1. Architecture Modeled

We model a 45nm architecture with a processor similar to an Alpha 21264, 64KB L1 I- and D-caches and a 2MB L2 cache. We estimate a nominal frequency of 6GHz with a supply voltage of 1V. We use the simplified version of the Alpha 21264 pipeline shown in Figure 1(a). In the figure, labeled boxes represent logical pipeline stages, which are composed of one or more physical pipeline stages. Unlabeled boxes show pipeline registers between logical stages. The pipeline registers between the multiple physical stages of some logical stages are not shown.

The Alpha 21264 pipeline has a logic depth of approximately 17FO4 per pipeline stage [23]. As per [20], we choose the setup and hold times to be 10% and 4%, respectively, of the nominal clock period. This gives us a nominal period of 18.8FO4 and a setup and hold times of 1.8FO4 and 0.8FO4, respectively. In some experiments, we scale the logic depth of the pipeline stages from 17FO4 to 6FO4; in all cases, we use the same absolute value of the setup and hold times.

We take the latencies of the different pipeline structures at 17FO4 from [23]. We follow the methodology in [21] in that, as the logic depth of stages decreases, we add extra pipeline stages to keep the total algorithmic work in the pipeline constant. Finally, we are assuming that, before variation, the pipeline stages are balanced. This represents the most unfavorable case for ReCycle.

The feedback path lengths are estimated based on the Alpha 21264 floorplan scaled down to 45nm. From ITRS projections, we use a wire delay of 371ps/mm [25].

6.2. ReCycle Hardware Overhead

The pipeline used in this paper (Figure 1(a)) has 23 physical pipeline stages organized into 13 logical ones. Consequently, as per Section 5.4, ReCycle needs the addition of 13 duplicate pipeline registers, 13 clock signal buffers connected to the duplicate registers, 36 inverter chains and multiplexers, one Donor Creation register and, optionally, one phase detector and predictor, and temperature sensors.

The area and power overhead of the duplicate pipeline registers, clock signal buffers, inverter chains, multiplexers, Donor Creation register, and temperature sensors is negligible. Specifically, we observe that the maximum clock skew T_{skew_max} that we need per stage is 50% of the nominal clock period. This corresponds to 0.5 $\times$ 18.8FO4 = 9.4FO4. Using 1FO4 $\approx$ 3FO1 from [22], and 1FO1 = 4ps at 45nm from [32], we have that T_{skew_max} = 112.8ps. This delay can be supplied by 28 basic inverters. Then, the multiplexer can be controlled by 5 bits. The resulting clock signal buffer with the inverter chain, multiplexer, and skew selector consumes negligible area and power. As a reference, a bigger buffer controlled by 16 bits at 800nm occupies just under 350μm $\times$ 150μm [13]. Linearly scaling the area to a 45nm design, and adding up the contributions of all the added buffers, we get a negligible area. Moreover, Chakraborty *et al*. [8] find the power overhead of TDBs to be minimal.

We can use a hardware-based phase detector and predictor like the one proposed by Sherwood *et al*. [40]. Using CACTI [44], we estimate that it adds $\approx$ 0.25% to the processor area.

6.3. Modeling Process Variation

We model a chip with four instances of the processor, L1 and L2 architecture described in Section 6.1 — although only one processor is being used. The chip's V_t and L_{eff} maps are generated using the model of Section 2.3. For V_t, we set μ=150mV at 100°C, and use empirical data from Friedberg *et al*. [19] to set σ/μ to 0.09. Following [27], we use equal contributions of the systematic and random components. Consequently, $\sigma_{sys}/\mu = \sigma_{ran}/\mu = \sqrt{\sigma^2/2}/\mu = 0.064$. Finally, since Friedberg *et al*. [19] observe that the range of spatial correlation is around half the length of the chip, we set the default ϕ to 0.5.

For L_{eff}, we use ITRS projections that set L_{eff}'s σ/μ design target to be 0.5 of V_t's σ/μ. Consequently, we use $\sigma/\mu = 0.045$ and $\sigma_{sys}/\mu = \sigma_{ran}/\mu = 0.032$. Knowing μ, σ, and ϕ, we generate chip-wide V_t and L_{eff} maps using the geoR statistical package [37] of R [35]. We use a resolution of 1M cells per chip, which corresponds to 256K cells for the processor and caches used. Each individual experiment is *repeated 200 times*, using 200 chips. Each chip has different V_t and L_{eff} maps generated with the parameters described. Finally, we ignore variation in wires, in agreement with current variation models [24].

6.4. Architecture Simulation Infrastructure

We measure the performance of the architecture of Section 6.1 with the SESC cycle-accurate execution-driven simulator [36]. We run all the SPEC2000 applications except 3 SPECint (eon, perlbmk, and bzip2) and 4 SPECfp (galgel, facerec, lucas, and fma3d) that fail to compile correctly. We evaluate each application for 0.6-1.0 billion instructions, after skipping several billion instructions due to initialization. The simulator is augmented with dynamic power models from Wattch [7] and CACTI [44].

7. Results

7.1. Timing Issues After Applying ReCycle

In any given pipeline, the loop with the longest average stage delay is the critical one, and limits ReCycle's ability to further reduce the pipeline period. In the rest of this paper, we use the term "stage in a loop" to refer to the combination of the loop's physical stage(s) and its feedback path(s).

Figure 7 shows the fraction of times that each of the loops in Figure 1(b) is critical for a batch of 200 chips. This figure demonstrates the interplay of several factors: the number of logical stages in a loop, the number of physical stages in each logical stage of the loop, and the relative number of feedback paths in the loop. A large number of logical stages in a loop induces a better averaging of stage delays, since the probability that all logical stages are slow is small. More physical stages per logical stage reduces the effectiveness of ReCycle. The reason is that, since all these physical stages share the same hardware structure, their critical paths are affected by the same values of the systematic component of variation. As a result, they contribute with similar delays to the loop. Finally, since wires are not subject to variation in our model and, in most cases, a stage with logic is slowed down due to a slow critical path, having relatively more feedback paths in a loop reduces its average delay.

The figure shows that there are two types of pipeline loops that are unlikely to be critical. One is very short loops, such as *iren*, *fpren*, *iissue*, *fpissue*, and *ialu*. These loops have two stages, including one feedback path. The latter is effective at reducing the average loop delay. The second type is long loops, such as *ldfwd*. This loop has 13 stages, which include 2 feedback paths. It is likely that it contains some fast stages that reduce the average stage delay.

On the other hand, medium-sized loops that include several physical stages in the same logical stage are often critical. They

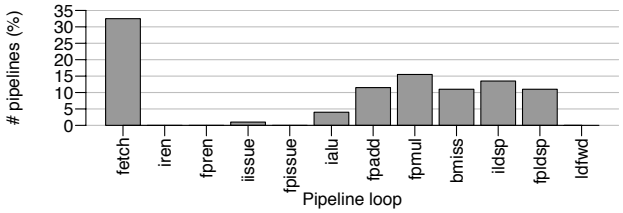


Figure 7: Histogram of critical pipeline loops.

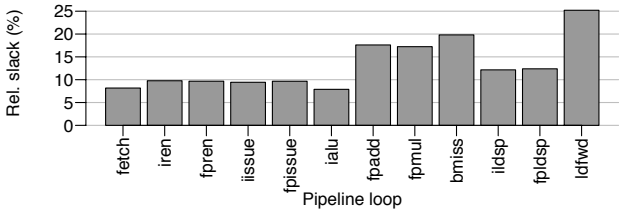


Figure 8: Average slack per stage in each loop. The data is shown relative to the stage delay of a no-variation pipeline.

include *fetch*, *fpmul* and others. In these loops, the feedback path only has modest impact at reducing the average delay, and the fact that multiple physical stages are highly correlated opens the door to unfavorable cases.

For a given pipeline, only one loop is critical, and the rest have unused timing slack. For the same experiment as in the previous figure, Figure 8 shows the average slack per stage in each loop. The data is shown relative to the stage delay of a no-variation pipeline.

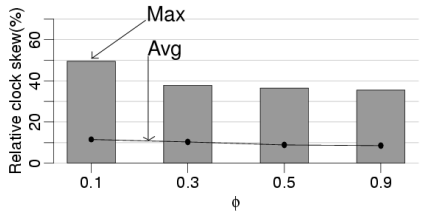
The data shows that, in general, the average slack per stage in a loop tends to increase with the number of stages making up the loop. This is because more stages tend to produce better averages and reduce the possibility of making the loop critical. We observe that, in the longest loop (*ldfwd*), we get an average slack per stage of 25%. The main exception to this trend is *fetch*, which has a low average slack even though it has 5 stages. The reason is that it is critical for the largest number of pipelines (Figure 7) and, therefore, has no slack in those cases.

Finally, we measure the average and maximum time skew that ReCycle inserts per pipeline register. We show this data as we change the range ϕ from its default 0.5 to higher and lower values (Figure 9(a)), and as we reduce the useful logic depth of the pipeline stages from the default 17FO4 to 6FO4 (Figure 9(b)). In both cases, we show the skews relative to the stage delay of a no-variation pipeline for the same logic depth of the stages.

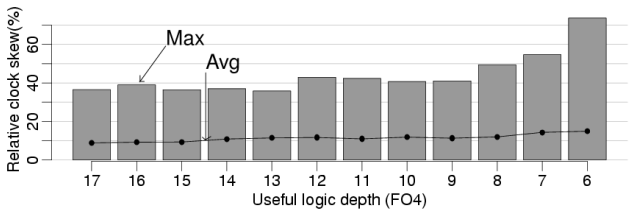
The average skew is a measure of the average stage unbalance in the pipeline loops. Figure 9(a) shows that the average skew increases as we reduce ϕ. This is because for low ϕ, even short loops observe large changes in systematic variation, which increase unbalance. Similarly, Figure 9(b) shows that the average skew increases as we decrease the logic depth. The reason is that, for shorter stages, the random component of the variation is more prominent, increasing the unbalance. Finally, both figures show that the maximum skews are much higher than the average ones. For example, for 17FO4 and ϕ= 0.1, the maximum skew reaches 0.5.

7.2. Frequency After Applying ReCycle

We now consider the impact of ReCycle on the pipeline frequency. We compare three environments, namely one without process variation (*NoVar*), one with process variation and no ReCycle (*Var*), and one with process variation and ReCycle (*ReCycle*). Figure 10 compares these environments for 17FO4 as we vary ϕ. All bars are normalized to the frequency of *Var* for ϕ=0.1.



(a) Skew versus ϕ



(b) Skew versus logic depth

Figure 9: Average and maximum time skew inserted by ReCycle per pipeline register. The skews are shown relative to the stage delay of a no-variation pipeline of the same logic depth.

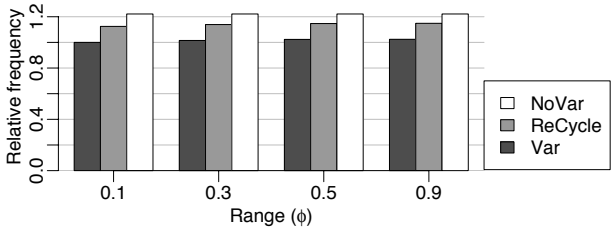


Figure 10: Pipeline frequency of the environments considered for different ϕ.

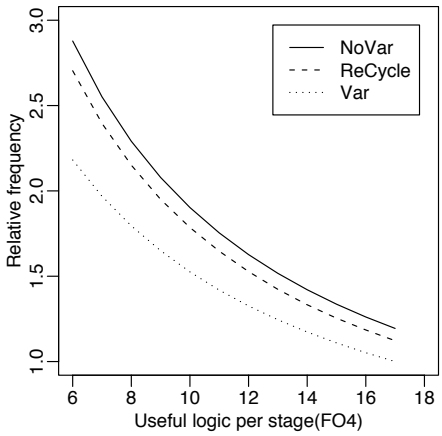


Figure 11: Pipeline frequency of the environments considered for different useful logic depths per pipeline stage.

The figure shows that, across different ϕ, the pipeline frequency of *NoVar* is 19-22% higher than that of *Var*. This would be the frequency gain if we could completely eliminate the effect of variation. On the other hand, the frequency of Recycle is about 12-13% higher than *Var*'s. This means that ReCycle is able to recover around 60% of the frequency losses due to variation. The figure also shows that frequencies are not very sensitive to the value of ϕ, possibly because many factors end up affecting the critical loop.

Figure 11 compares the three environments for ϕ=0.5 as we vary the useful logic depth per pipeline stage from 17FO4 to 6FO4. All

the curves shown in the figure are normalized to the frequency of *Var* for 17FO4.

The figure shows that, as we decrease the logic depth per stage, process variation hurts the frequency of a *Var* pipeline more and more. Indeed, while *NoVar*'s frequency is 19% higher than *Var*'s at 17FO4, it is 32% higher at 6FO4. This is because, with fewer gates per critical path for 6FO4, the random component of process variation does not average itself as much, creating more unbalance across stages and hurting *Var*. However, a pipeline with ReCycle is very resilient to variation, and tracks *NoVar* well. ReCycle performs relatively better as the logic depth per pipeline stage decreases. Specifically, ReCycle's frequency is 12% and 24% higher than *Var*'s for 17FO4 and 6FO4, respectively. This means that ReCycle recovers 63% and 75% of the losses due to variation for 17FO4 and 6FO4, respectively. The stage-delay averaging capability of ReCycle is very effective. Overall, ReCycle puts pipelines with variation back on the roadmap to scaling.

7.3. Adding Donor Stages

After ReCycle is applied, we can further improve the pipeline frequency by adding donor stages. Section 4.2 described the Donor algorithm that we use. In practice, every time that we add a donor stage to a loop, the loop typically ends up with the highest average slack per stage in the pipeline and another loop becomes critical. Recall that the Donor algorithm stops when we reach the power limit. We set the power limit to 30W per processor, which is the maximum power dissipated by any of our applications on a *NoVar* processor at the nominal frequency.

We have run the Donor algorithm statically (Section 4.2), using as representative workload the execution of all our applications, one at a time, and minimizing the impact on the geometric mean of the IPCs. It can be shown that, on average for the 200 pipelines considered, the Donor algorithm pushes up the frequency of ReCycle-enhanced pipelines by a further 24%, resulting in an average frequency that is now 36% higher than *Var*.

As an example, Figure 12(a) shows the frequency changes as we run the Donor algorithm on one representative pipeline instance. Each data point corresponds to the addition of one donor stage. We see that the frequency increases slowly until when we add the 9th donor stage; at that point, there is large frequency increase. This corresponds to the point when all the loops in the pipeline have been given donor stages — while the pipeline has 12 loops, some of them share a donor stage. After that, frequency increases are again small.

Since increasing the latency of a pipeline loop hurts IPC, not every step in frequency increase translates into a performance increase. Figure 12(b) shows the performance changes for the pipeline instance of Figure 12(a). If we focus on the curve that includes all applications, we see that the performance changes little with more donor stages until we add the 9th stage. At that point, the performance jumps up, even surpassing the performance of the pipeline without variation (*NoVar*). After that, additional stages improve performance slowly again. The figure also shows the curve if we had used only SPECint or SPECfp applications to profile the IPC changes.

Examining the data for the 200 pipelines analyzed, we observe several trends. First, as we add donor stages, performance decreases or stays flat for the first few steps and then starts increasing. Second, when each loop has at least one donor stage, we observe a significant performance boost. Third, the performance reaches a

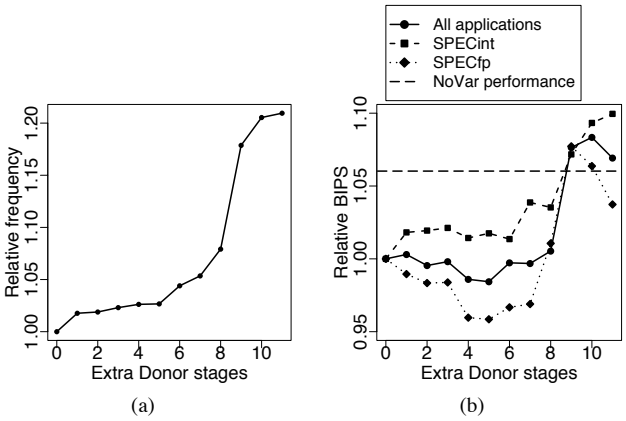


Figure 12: Impact of donor stages on frequency (a) and performance (b) in one representative pipeline instance.

maximum after a few steps and then starts decreasing. Finally, the optimal performance delivered by this technique is always higher than that of the no-variation pipeline.

To gain additional insight, we run the Donor algorithm with a single application at a time, and measure the performance gains. It is as if we were tuning the pipeline to run that single application. Figure 13 shows the number of donor stages required for the pipeline to match the performance of the no-variation pipeline. The figure shows a bar for each application and the geometric mean (last bar). For a given application, the bar shows the mean of the 200 pipelines, while the segment shows the range of values for individual pipelines. We see that, in the large majority of applications, the Donor algorithm needs to add 8-12 stages to match *NoVar*. This figure also shows that all applications are amenable to the Donor algorithm.

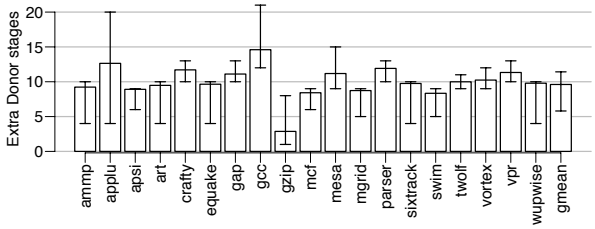


Figure 13: Number of donor stages needed to match the performance of the no-variation pipeline.

7.4. Overall Performance and Power Evaluation

We now compare the performance of *Var*, *NoVar*, ReCycle, ReCycle plus the static application of the Donor algorithm (*ReCycle+StDonor*), and ReCycle plus the dynamic application of the Donor algorithm (*ReCycle+DynDonor*). The latter is a limited dynamic environment, where we consider each whole application to be a single phase and, therefore, only rerun the Donor algorithm at the beginning of each application. Moreover, we assume that we know the average IPC impact of adding each donor stage from a previous profiling run. Modeling a more sophisticated dynamic environment will likely produce better results.

Figure 14 shows the performance of the five environments normalized to *Var*. The bars show the mean of the 200 pipelines, while the segments show the range for individual pipelines. Looking at average values, we see that ReCycle speeds up the applications

by 9% over *Var*. This is not enough to compensate the effect of variation, since *NoVar* is 14% faster than *Var*. However, *ReCycle+StDonor* and *ReCycle+DynDonor* more than recover the losses due to variation, since they are 15% and 16% faster than *Var*, respectively.

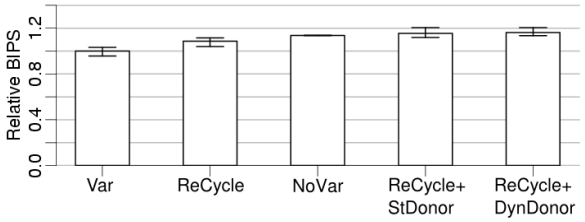


Figure 14: Performance of different environments.

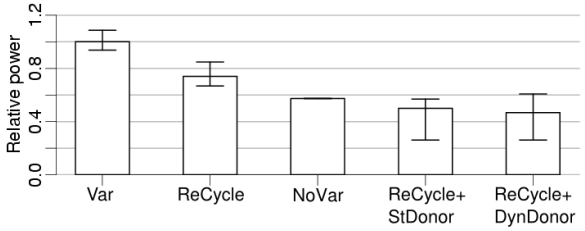


Figure 15: Dynamic power for constant performance.

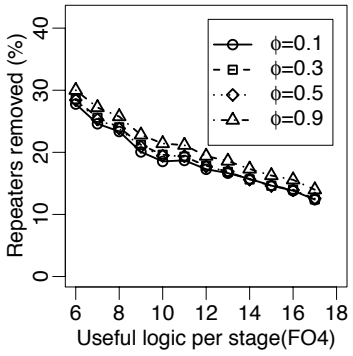


Figure 16: Fraction of repeaters eliminated by ReCycle.

ReCycle generates timing slack that we use to increase the frequency. Interestingly, we could use the slack generated by ReCycle to save power instead. Specifically, we could use dynamic voltage scaling to bring the frequency back to the *Var* level while reducing the operating voltage — saving dynamic power in the process. Similarly, we can do the same thing with the slack created by the Donor optimization, except that now we want to roll back the frequency until we get the same performance as *Var*.

The results of these experiments appear in Figure 15. The figure shows the dynamic power consumed by the environments of Figure 14 at *constant performance*. We see that ReCycle saves on average 26% of the *Var* dynamic power. Moreover, *ReCycle+StDonor* and *ReCycle+DynDonor* save 50% and 53%, respectively of the *Var* dynamic power. These are sizable power reductions.

7.5. Eliminating Repeaters

Section 4.3.1 proposed using ReCycle to push the slack of non-critical loops to their feedback paths and then consuming it by eliminating repeaters there — and saving power in the process. Figure 16 shows the percentage of repeaters in feedback paths that ReCycle eliminates for different values of the useful logic depth per pipeline stage and ϕ.

The figure shows that ReCycle removes 15-30% of the repeaters, depending on the logic depth per stage. The higher effectiveness corresponds to pipelines with less logic per stage. This is because, as we make the pipeline longer and stages shorter, we have more unbalance across stages. This results in non-critical loops having more slack. The bigger the slack is, the more repeaters we can remove. On the other hand, the value of ϕ has little effect. Overall, since $\approx$50% of the power in feedback paths is in repeaters [26], ReCycle can save $\approx$7.5-15% of the power in feedback paths.

8. Related Work

ReCycle is an architectural framework for pipelines that comprehensively performs cycle time stealing after fabrication (either statically at the manufacturer site or dynamically based on operating conditions) to tolerate process variation. The most related areas of research are those of clock skew optimization and adaptive pipelining.

Clock skew optimization has been well studied in the circuits community [4]. It has been applied both at design time and after fabrication to improve circuit timing margins. Fishburn [18] was the first to propose a linear programming formulation to find the optimal clock skews in a circuit.

Several works use clock skewing to address the problem of process variation. Most of them apply skewing to latch elements in the clock distribution network to compensate for the effects of process variation *on the clock path delays* themselves (e.g., [41]). For example, Itanium has buffers in the clock network that dynamically *deskew* the signal — i.e., ensure that the clock signal reaches all the parts of the processor with the same skew [14]. On the other hand, Liang and Brooks [31] use clock skewing to balance two pipeline stages. Specifically, they use cycle time stealing between the register file and execute stages and, with level-sensitive latches, between stages in the floating-point unit.

The only work that applies cycle time stealing in a systematic manner in a pipeline is that of Lee *et al*. [30]. They use it in the context of Razor to balance pipeline error rates. They neither apply it to process variation nor, more importantly, study the impact of pipeline structure such as pipeline depth or loop organization on the performance of cycle time stealing.

Adaptive pipelining techniques are related to our donor stage optimization. Koppanalil *et al*. [29] study the effect of dynamically merging pipeline stages to extend the frequency range of dynamic voltage scaling; they do not explicitly describe the implementation details of their scheme. Efthymiou *et al*. [16] use asynchronous design techniques to adaptively merge adjacent stages in an embedded, single-issue processor pipeline. Albonesi [2] proposes dynamically-varying functional unit latencies as an adaptive processing scheme, but he does not discuss the resulting scheduling complexities. Recently, Ozdemir *et al*. [34] address the issue of scheduling complexity for variable-access L1 cache by using additional load-bypass buffers. Finally, concurrently with our work, Liang and Brooks [31] propose inserting level-sensitive latches inside the FP unit that can be enabled after fabrication. If process variation is such that the unit does not meet timing, the latches are enabled, adding one extra cycle to the FP unit.

9. Conclusions

Process variation affects processor pipelines by exacerbating pipeline unbalance and reducing the attainable frequency. To toler-

ate variation, this paper proposed an architectural framework called *ReCycle* that comprehensively applies cycle time stealing — transferring the timing slack of the faster stages to the slower ones by skewing the clock arrival times to latching elements after fabrication. As a result, the pipeline can be clocked with a period close to the *average* stage delay rather than the longest one.

We showed that ReCycle increases the frequency of a pipeline without changing its structure or depth, or the speed of transistors. Such increase is relatively higher the deeper the pipeline is. Moreover, we proposed donor pipeline stages, which are empty stages added to the critical loop in the pipeline to "donate" slack to slow stages, enabling a higher pipeline frequency. We also used ReCycle to push the slack of non-critical pipeline loops to their feedback paths, which can then be consumed there to reduce wire power or to improve wire routability. Finally, ReCycle can also be used to salvage chips that would otherwise be rejected due to variation-induced hold-time failures.

On average for a 17FO4 pipeline, ReCycle increased the frequency by 12%, thereby recovering 63% of the frequency lost to variation, and speeding up our applications by 9%. Combining ReCycle and donor stages increased the frequency of the original pipeline by 36% and its performance by 15% on average. The resulting pipeline performed even better than one without process variation. Finally, ReCycle also saved 7-15% of the power in feedback paths for different pipeline depths.

References

[1] M. Agarwal, B. C. Paul, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *IEEE VLSI Test Symp.*, 2007.

[2] D. H. Albonesi. Dynamic IPC/clock rate optimization. In *International Symposium on Computer Architecture*, June 1998.

[3] C. Albrecht, B. Korte, J. Schietke, and J. Vygen. Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip. *Discrete Appl. Math.*, 123(1-3):103–127, 2002.

[4] K. Bernstein, K. M. Carrig, C. M. Durham, P. R. Hansen, D. Hogenmiller, E. J. Nowak, and N. J. Rohrer. *High Speed CMOS Design Styles*. Kluwer Academic Publishers, 1999.

[5] E. Borch, E. Tune, S. Manne, and J. S. Emer. Loose loops sink chips. In *International Symposium on High-Performance Computer Architecture*, February 2002.

[6] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, 2002.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[8] A. Chakraborty, K. Duraisami, A. Sathanur, P. Sithambaram, L. Benini, A. Macii, E. Macii, and M. Poncino. Dynamic thermal clock skew compensation using tunable delay buffers. In *International Symposium on Low Power Electronics and Design*, October 2006.

[9] T. Chen and S. Naffziger. Comparison of adaptive body bias (ABB) and adaptive supply voltage (ASV) for improving delay and leakage under the presence of process variation. *IEEE Trans. on VLSI Systems*, 11(5):888–899, October 2003.

[10] Z. Chishti and T. Vijaykumar. Wire delay is not a problem for SMT (in the near future). In *International Symposium on Computer Architecture*, June 2004.

[11] L. Cotten. Maximum rate pipelined systems. In *AFIPS Spring Joint Computing Conference*, 1969.

[12] N. Cressie. *Statistics for Spatial Data*. John Wiley & Sons, 1993.

[13] A. DeHon, T. Knight, Jr., and T. Simon. Automatic impedance control. In *ISSCC Digest of Technical Papers*, February 1993.

[14] U. Desai, S. Tam, R. Kim, J. Zhang, and S. Rusu. Itanium processor clock design. In *International Symposium on Physical Design*, April 2000.

[15] S. Dhar, D. Maksimovic, and B. Kranzen. Closed-loop adaptive voltage scaling controller for standard-cell ASICs. In *International Symposium on Low Power Electronics and Design*, August 2002.

[16] A. Efthymiou and J. D. Garside. Adaptive pipeline depth control for processor power-management. In *International Conference on Computer Design*, November 2002.

[17] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *International Symposium on Microarchitecture*, December 2003.

[18] J. P. Fishburn. Clock skew optimization. In *IEEE Trans. on Computers*, volume 39, July 1990.

[19] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos. Modeling within-die spatial correlation effects for process-design co-optimization. In *International Symposium on Quality Electronic Design*, March 2005.

[20] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon. High-performance microprocessor design. *Journal of Solid-State Circuits*, 33(5):676–686, May 1998.

[21] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *International Symposium on Computer Architecture*, May 2002.

[22] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4), April 2001.

[23] M. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. *International Symposium on Computer Architecture*, May 2002.

[24] E. Humenay, D. Tarjan, and K. Skadron. Impact of parameter variations on multicore chips. In *Workshop on Architectural Support for Gigascale Integration (ASGI)*, June 2006.

[25] International Technology Roadmap for Semiconductors (2005 Edition).

[26] P. Kapur, G. Chandra, and K. C. Saraswat. Power estimation in global interconnects and its reduction using a novel repeater optimization methodology. In *Design Automation Conference*, June 2002.

[27] T. Karnik, S. Borkar, and V. De. Probabilistic and variation-tolerant design: Key to continued Moore's law. In *TAU Workshop*, 2004.

[28] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[29] J. Koppanalil, P. Ramrakhyani, S. Desai, A. Vaidyanathan, and E. Rotenberg. A case for dynamic pipeline scaling. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2002.

[30] S. Lee, S. Das, T. Pham, T. Austin, D. Blaauw, , and T. Mudge. Reducing pipeline energy demands with local DVS and dynamic retiming. In *International Symposium on Low Power Electronics and Design*, pages 319–324, August 2004.

[31] X. Liang and D. Brooks. Mitigating the impact of process variations on CPU register file and execution units. In *International Symposium on Microarchitecture*, December 2006.

[32] B. Nikolic, L. Chang, and T.-J. King. Performance of deeply-scaled, power-constrained circuits. In *International Conference on Solid State Devices and Materials*, September 2003.

[33] M. Olivieri. Design of synchronous and asynchronous variable-latency pipelined multipliers. In *IEEE Transactions on Very Large Scale Integration Systems*, volume 9, April 2001.

[34] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou. Yield-aware cache architectures. In *International Symposium on Microarchitecture*, December 2006.

[35] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, 2005.

[36] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[37] P. Ribeiro Jr. and P. Diggle. geoR: a package for geostatistical analysis. *R-NEWS*, 1(2):14–18, June 2001.

[38] T. Sakurai and R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *Journal of Solid-State Circuits*, 25(2):584–594, 1990.

[39] T. Shanley. *The Unabridged Pentium-4*. Addison Wesley, July 2004.

[40] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture*, June 2003.

[41] M. Shoji. Elimination of process-dependent clock skew in CMOS VLSI. In *Journal of Solid State Circuits*, pages 875–880, 1986.

[42] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *International Symposium on Computer Architecture*, May 2002.

[43] A. Srivastava, D. Sylvester, and D. Blaauw. *Statistical Analysis and Optimization for VLSI: Timing and Power*. Springer, 2005.

[44] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report 2006/86, HP Laboratories, June 2006.

[45] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Journal of Solid-State Circuits*, 37(11):1396–1402, 2002.

[46] X. Vera, O. Ünsal, and A. González. X-pipe: An adaptive resilient microarchitecture for parameter variations. In *Workshop on Architectural Support for Gigascale Integration*, June 2006.

[47] J. Xiong, V. Zolotov, and L. He. Robust extraction of spatial correlation. In *International Symposium on Physical Design*, August 2006.